(1)

July 1991

M90-19

W. M. Farmer
J. D. Guttman
F. J. Thayer

IMPS: An Interactive Mathematical Proof System

CONTRACT SPONSOR MSR
CONTRACT NO. N/A
PROJECT NO. 9128A
DEPT. G117

MITRE

The MITRE Corporation
Bedford, Massachusetts

Abstract

IMPS is an Interactive Mathematical Proof System intended as a general purpose tool for formulating and applying mathematics in a familiar fashion. The logic of IMPS is based on a version of simple type theory with partial functions and subtypes. Mathematical specification and inference are performed relative to axiomatic theories, which can be related to one another via inclusion and theory interpretation. The system supports a natural style of inference based on *deduction graphs*, which are akin to derivations in a sequent calculus.

Acknowledgments

Several of the key ideas behind IMPS were originally developed by Dr. Leonard Monk on the Heuristics Research Project sponsored by The MITRE Corporation during 1984–87. Some of these ideas are described in [24].

This work was supported by the MITRE-Sponsored Research program under project 91280.

Table Of Contents

Section 1
Introduction

IMPS is an Interactive Mathematical Proof System, currently being developed at The MITRE Corporation, that is intended to support the axiomatic method. The IMPS user can specify axiomatic theories, interactively prove theorems in them, and relate one theory to another via both inclusion and theory interpretation. The guiding goal of IMPS is to provide strong computational support for rigorous mathematical reasoning in a framework that closely models standard mathematical practice. Clarity and flexibility of expression are thus valued over raw deductive power.

The general goals of IMPS fall into five areas:

- *Logic.* The logic of the system should provide a precise, flexible framework in which to express many kinds of mathematical specification and inference. It should be *easily accessible to the user* by utilizing standard syntactic and semantic ideas. And, it should allow the user to formulate mathematical concepts and arguments in a natural and direct manner. In particular, the logic should not be based on restrictive or unusual methods.

- *Proofs.* The system should support the interactive development of intelligible formal proofs. There should be essentially no structural difference between partial and complete proofs. Proofs should be encoded by internal data structures that can be manipulated and analyzed by software.

- *Computational support.* The system should provide several kinds of computational support, including syntax checking, expression simplification, and various kinds of assistance for building and presenting formal proofs.

- *Reusability.* The user should be able to reuse previously formulated expressions, languages, theories, and arguments. There should be support for constructing new theories from old theories either directly or via theory interpretations, and it should be possible to develop abstract mathematical results and then use them in a variety of more concrete contexts.

- *Human-machine interaction.* It is essential that the system has a congenial interface that gives the user wide access to the system while protecting the user from making unsound inferences. The user should be able to tailor machine deduction to his own needs, and machine calculations should generally be performed in less than a minute.

This paper presents an overview of the IMPS system. The next section, Section 2, describes the logic of IMPS which is a based on a version of simple type theory with partial functions and subtypes. Section 3 discusses the role axiomatic theories play in IMPS and explains how they can be extended by definitions and be related to one another by theory interpretations. The theorem proving system of IMPS is described in Section 4; it supports a natural style of inference based on *deduction graphs*, which are akin to derivations in a sequent calculus. Section 5 describes some of the ways the proof process in IMPS is driven by information contained in the axioms and theorems of a theory. Applications and the implementation are briefly discussed in Sections 6 and 7, respectively. Section 8 contains, as an example, a proof that the real numbers satisfy the Archimedean property. Finally, a brief conclusion is given in Section 9.

Section 2
Logic

The logic[1] of IMPS is called LUTINS[2], a Logic of Undefined Terms for Inference in a Natural Style. LUTINS is a conceptually simple implementation of higher-order predicate logic that closely conforms to mathematical practice. Partial functions are dealt with directly, and consequently, terms may be nondenoting. The logic, however, is bivalent; formulas are always defined.

LUTINS is derived from the formal system **PF*** [10], which in turn is derived from the formal system **PF** [9]. **PF** is a version of Church's simple theory of types in which functions may be partial, and **PF*** is a multi-sorted, multi-variate simple type theory with partial functions, subtypes, and definite description operators. LUTINS is essentially **PF*** plus a number of convenient expression constructors, which are discussed below. It is shown in [9] and [10] that **PF** and **PF***, respectively, are complete with respect to a Henkin-style general models *semantics* [18]. The formal semantics of LUTINS is straightforwardly derived from the (standard models) semantics of **PF***. (See [17] for a detailed description of the syntax and semantics of LUTINS.)

2.1 Higher-Order Functions and Types

Higher-order logic (or type theory) was developed in the early part of this century to serve as a foundation for mathematics, but lost its popularity as a foundation for mathematics in the 1930's with the rise of set theory and first-order logic. Higher-order logic emphasizes the role of functions, in contrast

[1]By a *logic*, we mean in effect a function. Given a particular vocabulary, or set of (nonlogical) constants, the logic yields a triple consisting of a formal language $\mathcal{L}$, a class of models $\mathcal{A}$ for the language, and a satisfaction relation $\models$ between models and formulas. The function is normally determined by the syntax and semantics of a set of logical constants for the logic.

The satisfaction relation determines a *consequence* relation between sets of formulas and individual formulas. A formula P is a consequence of a set of formulas S if $\mathcal{A} \models P$ holds whenever $\mathcal{A} \models Q$ holds for every $Q \in S$.

When we speak of a theory, we mean in essence a language together with a set of axioms. A formula is a theorem of the theory if it is a consequence of the axioms.

[2]Pronounced as the word in French.

to set theory, which emphasizes the role of sets. In type theory, functions may be quantified and may take other functions as arguments. In order to avoid circularity, functions are organized according to a type hierarchy.

Type theory has a uniform syntax; it is based on familiar notions; and it is highly expressive. The use of λ-notation allows functions to be specified succinctly. Since type theory contains second-order logic, there are many things that can be expressed in it which cannot be directly expressed in first-order logic. For example, the induction principle for the natural numbers can be expressed completely and naturally by a single second-order formula. See [3] and [30] for discussion on the expressive power of second-order logic relative to first-order logic.

The type hierarchy of LUTINS consists of base types and function types. Let $\mathcal{L}$ be a language in LUTINS. The *base types* of $\mathcal{L}$ are $\mathbf{prop}, \mathbf{ind}_1, \ldots, \mathbf{ind}_m$ where $m \geq 1$. $\mathbf{prop}$ is the type of propositions and each $\mathbf{ind}_i$ is a type of individuals. The *function types* of $\mathcal{L}$ are inductively defined from its base types: if $\alpha_1, \ldots, \alpha_n, \alpha_{n+1}$ are (base or function) types where $n \geq 1$, then $\alpha_1, \ldots, \alpha_n \rightarrow \alpha_{n+1}$ is a function type. Since m and n may be strictly greater than 1, the type structure is "multi-sorted" and "multi-variate," respectively.

A higher-order logic with this sort of type hierarchy is called a *simple type theory*. The automatic theorem proving system TPS developed at CMU [1] and the proof development system HOL developed at the University of Cambridge [15] are both based on simple type theories. However, in these systems function types contain only total functions, while in LUTINS, some types may contain partial functions. These are the *types of kind* $\mathbf{ind}$. We say that a type α is of *kind* $\mathbf{ind}$ if $\alpha = \mathbf{ind}_i$ for some $i \geq 1$ or $\alpha = \alpha_1, \ldots, \alpha_n \rightarrow \alpha_{n+1}$ and α_{n+1} is of kind $\mathbf{ind}$. Otherwise, we say that α is of *kind* $\mathbf{prop}$.

Every formal expression in LUTINS has a unique type. The type of an expression serves both a semantic and syntactic role: An expression denotes an object in the denotation of its type (if the expression is defined), and the syntactic well-formedness of an expression is determined on the basis of the types of its components. An expression is said to be of *kind* $\mathbf{ind}$ [$\mathbf{prop}$] if its type is of kind $\mathbf{ind}$ [$\mathbf{prop}$]. Expressions of kind $\mathbf{ind}$ are used to describe mathematical objects; they may be undefined. Expressions of kind $\mathbf{prop}$ are primarily used in making assertions about mathematical objects; they are always defined.

2.2 Partial Functions

One of the primary distinguishing characteristics of LUTINS is its direct approach to specifying and reasoning about partial functions (i.e., functions which are not necessarily defined on all values). Partial functions are ubiquitous in both mathematics and computer science. If a term is constructed from simpler expressions by the application of an expression denoting a partial function f to an expression denoting a value a which is outside the domain of f, then the term itself has no natural denotation. Such a term would violate the *existence assumption* of classical logic, which says that terms always have a denotation. Thus a direct handling of partial functions can only lie outside of classical logic.[3]

The semantics of LUTINS is based on five principles:

(1) Expressions of kind ind may denote partial functions.

(2) Expressions of type prop always denote a standard truth value.

(3) Variables, constants, and λ-expressions always have a denotation.

(4) An application of kind ind is undefined if its function or any of its arguments is undefined.

(5) An application of type prop is false if any of its arguments is undefined.

As a consequence of these principles, expressions of kind ind may be nondenoting, but expressions of kind prop must be denoting. Hence the semantics of LUTINS allows partial functions without sacrificing bivalent logic. We have chosen this approach for dealing with partial functions because it causes minimal disruption to the patterns in reasoning familiar from classical logic and standard mathematical practice. (For a detailed discussion of the various ways of handling partial functions in predicate logic, see [9].)

[3]However, since the graph of a function (partial or total) can always be represented as a relation, the problem of nondenoting terms can in theory be easily avoided—at the cost of using unwieldy, verbose expressions. Hence, *if pragmatic concerns are not important*, classical logic is perfectly adequate for dealing with partial functions.

2.3 Constructors

The expressions of a language of LUTINS are constructed from variables and constants by applying *constructors*. Constructors serve as "logical constants" that are available in every language. LUTINS has approximately 20 constructors. (**PF** and **PF*** have only two constructors, application and λ-abstraction.) Logically, the most basic constructors are **`apply-operator`**, **`lambda`**, **`iota`**, and **`equality`**; in principle every expression of LUTINS could be built from these four.[4] The other constructors serve to provide economy of expression.

There is a full set of constructors for predicate logic: constants for true and false, propositional connectives, quantifiers, and equality. LUTINS also has a definite description operator **`iota`**, an if-then-else operator **`if`**, and some definedness constructors such as **`is-defined`** (denoted by the postfix symbol $\downarrow$) and **`is-defined-in-sort`** (infix $\downarrow$). Although a few constructors (such as **`implies`** (infix $\supset$) and **`not`** ($\neg$)) correspond to genuine functions, most constructors do not. For example, the constructors **`and`** (infix $\wedge$) and **`or`** (infix $\vee$) are applicable to any number of formulas (i.e., expressions of type **`prop`**). The constructor **`if`** is nonstrict in its second and third arguments, and several constructors bind variables, including **`forall`**, **`forsome`**, and **`iota`**, in addition to the basic variable-binding constructor **`lambda`**.

`iota`, the definite description operator of LUTINS, is a constructor that cannot be easily imitated in other logics. Using this constructor, one can create a term of the form $\mathrm{I}x \,.\, P(x)$, where P is a predicate, which denotes the unique element described by P. More precisely, $\mathrm{I}x \,.\, P(x)$ denotes the unique x that satisfies P if there is such an x and is undefined otherwise. In addition to being quite natural, this kind of definite description operator is very useful for specifying (partial) functions. For example, ordinary division (which is undefined whenever its second argument is 0) can be defined from the times function $*$ by a λ-expression of the form

$$\lambda x, y \,.\, \mathrm{I}z \,.\, x * z = y.$$

In logics in which terms always have a denotation, there is no completely

[4]Throughout this paper, constructors will be denoted using traditional symbology. For example, **`lambda`** and **`iota`** are denoted, respectively, by the variable-binding symbols λ and I; **`equality`** is denoted by the usual infix symbol $=$; and **`apply-operator`** is denoted implicitly by the standard notation of function application.

satisfactory way to formalize a definite description operator. This is because a definite description term $\mathrm{I}x \,.\, P(x)$ must always have a denotation, even when there is no unique element satisfying P.

The IMPS implementation allows one to create macro/abbreviations called *quasi-constructors* which are defined in terms of the ordinary constructors. For example, the quasi-constructor **quasi-equality** (infix $\simeq$) is defined as follows:

$$e_1 \simeq e_2 \;\equiv\; (e_1\!\downarrow \vee e_2\!\downarrow) \supset e_1 = e_2.$$

Depending on the choice of the user, a quasi-constructor can be used in IMPS in two different modes: as a device for constructing expressions with a common form or as an ordinary constructor. The first mode is needed for proving basic theorems about quasi-constructors, while the second mode effectively gives the user a logic with a richer set of constructors. Quasi-constructors can be especially useful for formulating generic theories (e.g., a theory of finite sequences) and special-purpose logics within IMPS.

Constructors and quasi-constructors are polymorphic in the sense that they can be applied to expressions of several different types. For instance, the constructor `if` can take any three expressions as arguments as long as the type of the first expression is `prop` and the second and third expressions are of the same type.

2.4 Sorts

Superimposed on the type hierarchy of LUTINS is a system of subtypes. We call types and subtypes jointly *sorts*. The sort hierarchy consists of atomic sorts and compound sorts. Let $\mathcal{L}$ be a language in LUTINS. $\mathcal{L}$ contains a set of *atomic sorts* which includes the base types of $\mathcal{L}$. The *compound sorts* of $\mathcal{L}$ are inductively defined from the atomic sorts of $\mathcal{L}$ in the same way that function types of $\mathcal{L}$ are defined from the base types of $\mathcal{L}$. Every atomic sort is assigned an *enclosing sort.* (The enclosing sort of a base type is itself.) The assignment of enclosing sorts determines a partial order $\preceq$ with the following properties:

- $\alpha \preceq \beta$ whenever β is the enclosing sort of α.
- $\alpha_1, \ldots, \alpha_n \rightarrow \alpha_{n+1} \preceq \beta_1, \ldots, \beta_n \rightarrow \beta_{n+1}$ whenever $\alpha_i \preceq \beta_i$ for all i with $1 \leq i \leq n+1$.

- A sort is a maximal element in $\preceq$ iff it is a type.
- For all sorts α, there is a unique type β, called the *type* of α, such that $\alpha \preceq \beta$.
- The type of an atomic sort is of kind **prop** iff the atomic sort is itself **prop**.

A sort is said to be of kind ind [**prop**] if its type is of kind ind [**prop**]. If two sorts have the same type, then that type is clearly an upper bound for them in $\preceq$. Moreover, since each atomic sort has a single enclosing sort, an inductive argument shows that any two sorts of the same type have a least upper bound.

A sort denotes a subset of the denotation of its type. Hence sorts may overlap, which is very convenient for formalizing mathematics. (The overlapping of sorts has been dubbed *inclusion polymorphism*[4].) Since a partial function from a set A to a set B is also a partial function from any superset of A to any superset of B, compound sorts of kind ind have a very elegant semantics: The denotation of $\alpha = \alpha_1, \ldots, \alpha_n \rightarrow \alpha_{n+1}$ of type β of kind ind is the set of partial (and total) functions f of type β such that $f(a_1, \ldots, a_n)$ is undefined whenever at least one of its arguments a_i lies outside the denotation of α_i. (The semantics for compound sorts of kind **prop** is similar but less elegant.)

Sorts serve two main purposes. First, they help to specify the value of an expression. Every expression is assigned a sort on the basis of its syntax. If an expression is defined, it denotes an object in the denotation of its sort. Second, sorts are used to restrict the application of binding constructors. For example, if α is a sort of type β, then a formula of the form

$$\forall x : \alpha \,.\, P(x)$$

is equivalent to the formula

$$\forall y : \beta \,.\, (y \downarrow \alpha) \supset P(y).$$

Sorts are not directly used for determining the well-formedness of expressions. Thus, if f and a are expressions of sorts $\alpha \rightarrow \beta$ and α', respectively, then the application $f(a)$ is well-formed provided only that α and α' have the same type.

As a simple illustration of the effectiveness of this subtyping mechanism, consider the language of our theory of real numbers, `h-o-real-arithmetic`, in which we stipulate **N** is enclosed by **Z**, which is enclosed by **Q**, which is enclosed by **R**, which is enclosed by the base type $\mathtt{ind}_1$. So $\mathbf{N} \rightarrow \mathbf{R}$ denotes the set of all partial functions from the natural numbers to the real numbers. This set of functions is a subset of the denotation of $\mathtt{ind}_1 \rightarrow \mathtt{ind}_1$. A function constant specified to be of sort $\mathbf{R} \rightarrow \mathbf{R}$ would automatically be applicable to expressions of sort **N**. Similarly, a function constant f declared to be of sort $\mathbf{N} \rightarrow \mathbf{N}$ would automatically be applicable to expressions of sort **R**, but an application $f(a)$ would only be defined when a denoted a member of the natural numbers. It is important to observe that a subtyping mechanism of this kind would be quite awkward in a logic having only total functions.

2.5 Summary

LUTINS is a multi-sorted, multi-variate higher-order predicate logic with partial functions and subtypes. Like other versions of simple type theory, it is highly expressive. It has strong support for specifying and reasoning about functions: λ-notation, partial functions, a true definite description operator, and full quantification over functions. Its type hierarchy and sort mechanism are convenient and natural for developing many different kinds of mathematics. Although LUTINS contains no polymorphism in the sense of variables over types, polymorphism is achieved through the use of constructors and quasi-constructors, sorts, and theory interpretations (see Subsection 3.2).

Perhaps most importantly, the intuition behind LUTINS closely corresponds to the intuition used in everyday mathematics. The logical principles employed by LUTINS are derived from classical predicate logic and standard mathematical practice. This puts it in contrast to some other higher-order logics, such as Martin-Löf's constructive type theory [23], the Coquand-Huet Calculus of Constructions [6], and the logic of the Nupri proof development system [5]. These logics—which are constructive as well as higher order—employ rich polymorphic type structures that incorporate the "propositions as types" isomorphism (see [21]). They are a significant departure from standard mathematical practice. Moreover, their type structures achieve a high level of polymorphism at the cost of increased semantic complexity.

Section 3
Theories

IMPS is a platform for rigorous mathematical reasoning based on the axiomatic method. The system allows users to specify axiomatic theories, to prove theorems within them, and to relate one theory to another via inclusion and theory interpretation. Mathematically, a theory in IMPS consists of a language and a set of axioms. At the implementation level, however, theories contain additional structure which tabulates or encodes procedurally this axiomatic information to facilitate various kinds of low-level reasoning within theories. The three most important examples are:

- Theory-specific algebraic simplification, for instance, simplification of polynomials when a theory contains the structure of a ring or field.
- Deciding satisfiability of linear inequalities, for instance, when a theory contains the structure of an ordered ring or field.
- Exploiting information about the domains and ranges of functions to infer whether terms are defined or undefined.

A theory is constructed from a (possibly empty) set of subtheories, a language, and a set of axioms. Theories are related to each other in two ways: one theory can be the *subtheory* of another, and one theory can be *interpreted* in another by a theory interpretation. A theory may be enriched via the definition of new atomic sorts and constants and via the installation of theorems. Definitions and theory interpretations are discussed below in the next two subsections. Several examples of theories are discussed in the last section.

3.1 Definitions

A theory in IMPS may be enriched by defining new sorts and constants. For example, the functions **min** and **max** from pairs of reals to reals, and the limit operator from sequences of reals to reals, are all defined constants in our standard theory of the real numbers. Functions may also be defined by recursion, using the general mechanism of fixed-point inductive definition analyzed by Moschovakis [25, 26]. Mutually recursive definitions are permitted. Simple examples of recursively defined operators include the $\sum$ and $\prod$ operators for summing and multiplying finite sequences.

IMPS supports four kinds of definitions: atomic sort definitions, constant definitions, recursive function definitions, and recursive predicate definitions. In the following let $\mathcal{T}$ be an arbitrary theory.

Atomic sort definitions are used to define new atomic sorts from nonempty unary predicates. An *atomic sort definition* for $\mathcal{T}$ is a pair $\delta = (n, P)$ where n is a symbol intended to be the name of a new atomic sort of $\mathcal{T}$ and P is unary predicate in $\mathcal{T}$ intended to specify the extension of the new sort. δ can be installed in $\mathcal{T}$ only if (1) n is not the name of any current sort of $\mathcal{T}$ or of a theory for which $\mathcal{T}$ is a structural subtheory, and (2) the formula $\exists x . P(x)$ is known to be a theorem of $\mathcal{T}$. When δ is installed in $\mathcal{T}$, a new atomic sort with the name n is added to the language of $\mathcal{T}$, and a new axiom is added to $\mathcal{T}$ which says that the new sort and P are coextensional.

Constant definitions are used to define new constants from defined expressions. A *constant definition* for $\mathcal{T}$ is a pair $\delta = (n, e)$ where n is a symbol intended to be the name of a new constant of $\mathcal{T}$ and e is an expression in the language of $\mathcal{T}$ intended to specify the value of the new constant. δ can be installed in $\mathcal{T}$ only if (1) n is not the name of any current constant of $\mathcal{T}$ or of a theory for which $\mathcal{T}$ is a structural subtheory and (2) the formula $e\downarrow$ is known to be a theorem of $\mathcal{T}$. When δ is installed in $\mathcal{T}$, a new constant c of the same sort as e with the name n is added to the language of $\mathcal{T}$, and a new axiom $c = e$ is added to $\mathcal{T}$.

Recursive function definitions are used to define one or more functions by (mutual) recursion. They are essentially an implementation of the approach to recursive definitions presented by Y. Moschovakis in [26]. A *recursive definition* for $\mathcal{T}$ is a pair $\delta = ([n_1, \ldots, n_k], [F_1, \ldots, F_k])$ where $k \geq 1$, $[n_1, \ldots, n_k]$ is a list of distinct symbols intended to be the names of a list of k new constants, and $[F_1, \ldots, F_k]$ is a list of functionals of kind $\mathtt{ind}$ in $\mathcal{T}$ intended to specify as a system the values of the new constants. δ can be installed in $\mathcal{T}$ only if (1) each n_i is not the name of any current constant of $\mathcal{T}$ or of a theory for which $\mathcal{T}$ is a structural subtheory, and (2) each functional F_i is known to be monotone with respect to the order $\sqsubseteq$ on partial functions defined by: $g \sqsubseteq g'$ iff g' is an extension of g (i.e., $g(a_1, \ldots, a_m) = g'(a_1, \ldots, a_m)$ for all m-tuples $\langle a_1, \ldots, a_m \rangle$ in the domain of g). When δ is installed in $\mathcal{T}$,

k new constants $f_1, \ldots, f_k$ with names $n_1, \ldots, n_k$, respectively, are added to the language of $\mathcal{T}$, and a new axiom is added to $\mathcal{T}$ which says that $f_i = F_i(f_1, \ldots, f_k)$ for each i with $1 \leq i \leq k$ and $[f_1, \ldots, f_k]$ is the minimum solution of $[F_1, \ldots, F_k]$ (with respect to $\sqsubseteq$).

This approach to recursive definitions is very natural in IMPS because expressions of kind ind are allowed to denote partial functions. Notice that there is no requirement that the functions defined by a recursive definition be total. In a logic in which functions must be total, a list of functionals can be a legitimate recursive definition only if it has a solution composed entirely of *total* functions. This is a difficult condition for a machine to check, especially when $k > 1$. Of course, in IMPS there is no need for a recursive definition to satisfy this condition since a recursive definition is legitimate as long as the defining functionals are monotone. IMPS has an automatic syntactic check for monotonicity that succeeds for many common recursive function definitions.

Recursive predicate definitions are used to define one or more predicates by (mutual) recursion. They are implemented in essentially the same way as recursive function definitions using the order $\subseteq$ on predicates defined by: $q \subseteq q'$ if q' includes q (i.e., $q(a_1, \ldots, a_m) \supset q'(a_1, \ldots, a_m)$ for all m-tuples $\langle a_1, \ldots, a_m \rangle$ in the common domain of q and q'). This approach is based on the classic theory of positive inductive definitions (see [25]). As with recursive function definitions, there is an automatic syntactic check for monotonicity that succeeds for most typical recursive predicate definitions.

3.2 Theory Interpretations

One of the chief virtues of the axiomatic method is that the theorems of a theory can be "transported" to any specialization of the theory. A theory interpretation is a syntactic device for translating the language of a source theory to the language of a target theory that has the property that the image of a theorem of the source theory is always a theorem of the target theory. It then follows that any theorem proved in the source theory translates to a theorem in the target theory. We use this method in a variety of ways (which are described below) to reuse mathematical results from abstract mathematical theories.

Theory interpretations are constructed in IMPS by giving an interpretation of the sorts and constants of the language of the source theory; this is the standard approach that is usually seen in logic textbooks (e.g., see [7] and

[31]).[5] We give below a summary of theory interpretations in IMPS; a detailed description of theory interpretations for $\mathbf{PF}^*$ is given in [10].

Let $\mathcal{T}$ and $\mathcal{T}'$ be theories over languages $\mathcal{L}$ and $\mathcal{L}'$, respectively. A *translation from $\mathcal{T}$ to $\mathcal{T}'$* is a pair (μ, ν), where μ is a mapping from the sorts of $\mathcal{L}$ to the sorts of $\mathcal{L}'$ and ν is a mapping from the constants of $\mathcal{L}$ to the expressions of $\mathcal{L}'$, such that:

(1) $\mu(\mathbf{prop}) = \mathbf{prop}$.

(2) For each sort α of $\mathcal{L}$, α and $\mu(\alpha)$ are of the same kind.

(3) If α is a sort of $\mathcal{L}$ with type β, then $\mu(\alpha)$ and $\mu(\beta)$ have the same type.

(4) If c is a constant of $\mathcal{L}$ of sort α, then the type of $\nu(c)$ is the type of $\mu(\alpha)$.

There is a canonical extension $\bar{\nu}$ of ν which maps expressions of $\mathcal{L}$ to expressions of $\mathcal{L}'$.

Let $\Psi = (\mu, \nu)$ be a translation from $\mathcal{T}$ to $\mathcal{T}'$. An *obligation* of Ψ is a formula $\bar{\nu}(\varphi)$ where φ is either:

(1) a (nonlogical) axiom of $\mathcal{T}$;

(2) a formula asserting that a particular atomic sort of $\mathcal{L}$ is a subset of its enclosing sort; or

(3) a formula asserting that a particular constant of $\mathcal{L}$ is a (defined) member of its sort.

By a theorem called the *theory interpretation theorem* (see [10]), Ψ is a *theory interpretation* from $\mathcal{T}$ to $\mathcal{T}'$ if each of its obligations is a theorem of $\mathcal{T}'$.

Theory interpretations are used extensively in IMPS in a variety of ways. The following are brief descriptions of the most important ways they are used.

[5]Although the theory interpretations available in IMPS are very general in nature, we shall restrict our attention to a subclass of theory interpretations which are especially easy to describe. In the more general case, the image of a sort under the interpretation may be a unary predicate, representing a subset of some sort of the target theory, rather than a sort of the target theory, as it is in the case described here. The more general version is somewhat cumbersome to describe.

for every $a, b : \mathbf{U}, m : \mathbf{Z}$ implication

- conjunction
  - o $not(a = o_{\mathbf{U}})$
  - o $not(b = o_{\mathbf{U}})$
  - o $1 \leq m$
- $(a \oplus b)^m = \sum_{k=0}^{m} *_{ext}(comb(m, k), a^{m-k} \otimes b^k)$.

Figure 1: The Binomial Theorem in Commutative Rings

Theorem reuse Mathematicians want to be able to formulate a result in the most general axiomatic framework that good taste and ease of comprehension allow. One major advantage of this approach is that a result proved in an abstract theory holds in all contexts that have the same structure as the abstract theory. In IMPS, theory interpretations are used foremost as a mechanism for realizing this advantage: theorems proved in abstract theories can be transported via a theory interpretation to all appropriate concrete structures. For instance, the binomial theorem may be proved in a (suitably formulated) theory of commutative rings (see Figure 1).[6] Because the real numbers form a commutative ring, we can define a theory interpretation from the commutative ring theory to a theory of the reals. As a consequence, we can then "install" the usual binomial theorem for the real numbers.

Automatic application of theorems Theorems can be automatically applied in IMPS in two ways: (1) as macetes (see Subsection 5.3) and (2) as rewrite rules (see Subsection 5.2). Theorems can be applied both inside and outside of their home theories. A theorem is applied within a theory $\mathcal{T}$ which is outside of its home theory $\mathcal{H}$ by, in effect, transporting the theorem from $\mathcal{H}$ to $\mathcal{T}$ and then applying the new theorem directly within $\mathcal{T}$. The theorem is transported by a theory interpretation that is either selected or constructed automatically by the system.

[6]In this formulation, $\mathbf{U}$ is the underlying sort of ring elements, $o_{\mathbf{U}}$ is the additive identity of the ring, $\oplus$ is ring addition and $\otimes$ is ring multiplication. The operation $*_{ext}$ multiplies an integer by a ring element, and means repeated ring addition, while exponentiation means repeated ring multiplication. Figure 1 is printed exactly as formatted by the the TeX presentation facility of IMPS. Various switches are available, for instance to cause connectives to be printed in-line with the usual logical symbols instead of being written as words with subexpressions presented in itemized format.

Polymorphic operators As we noted in Subsection 2.3, constructors and quasi-constructors are polymorphic in the sense that they can be applied to expressions of several different types. This sort of polymorphism is not very useful unless we have results about constructors and quasi-constructors that could be used in proofs regardless of the actual types that are involved. For constructors, most of these "generic" results are coded in the form of rules, as described in Subsection 4.2. Since quasi-constructors, unlike constructors, can be introduced by IMPS users, it is imperative that there is some way to prove generic results about quasi-constructors. This can be done by proving theorems about quasi-constructors in a theory of generic types, and then transporting these results as needed to theories where the quasi-constructor is used. For example, consider the quasi-constructor `composition` (infix $\circ$) defined as follows, for expressions f and g of type $\beta \rightarrow \gamma$ and $\alpha \rightarrow \beta$, respectively:

$$f \circ g \equiv \lambda x : \alpha \,.\, f(g(x)).$$

The basic properties about `composition`, such as associativity, can be proved in a generic theory having four base types but no constants, axioms, or other atomic sorts.

Symmetry and duality proofs Theory interpretations can be used to formalize certain kinds of arguments involving symmetry and duality. For example, suppose we have proved a theorem in some theory and have noticed that some other conjecture follows from this theorem "by symmetry." This notion of symmetry can frequently be made precise by creating a theory interpretation from the theory to itself which translates the theorem to the conjecture. As an illustration, let $\mathcal{T}$ be a theory of groups where $*$ is a binary constant denoting group multiplication. Then the translation from $\mathcal{T}$ to $\mathcal{T}$ which takes $*$ to $\lambda x, y \,.\, y * x$ and holds everything else fixed maps the left cancellation law $x * y = x * z \supset y = z$ to the right cancellation law $y * x = z * x \supset y = z$. Since this translation is in fact a theory interpretation, we need only prove the left cancellation law to show that both cancellation laws are theorems of $\mathcal{T}$.

Parametric theories As argued by Goguen (e.g., in [13] and [14]), a flexible notion of *parametric theory* can be obtained with the use of ordinary theories and theory interpretations. The key idea is that the primitives of a subtheory of a theory are a collection of parameters which can be instantiated as a group via a theory interpretation. For example, consider a generic theory $\mathcal{T}$ of graphs which contains a subtheory $\mathcal{T}'$ of abstract nodes and edges, and another theory $\mathcal{U}$ containing graphs with a concrete representation. The general results about graphs in $\mathcal{T}$ can be imported into $\mathcal{U}$ by creating a theory interpretation Ψ from $\mathcal{T}'$ to $\mathcal{U}$ and then lifting Ψ, in a completely mechanical way, to a theory interpretation of $\mathcal{T}$ to a definitional extension of $\mathcal{U}$. This use of theory interpretations has been implemented in OBJ3 as well as IMPS. (For a detailed description of this technique, see [8].)

Relative consistency If there is a theory interpretation from a theory $\mathcal{T}$ to a theory $\mathcal{T}'$, then $\mathcal{T}$ is consistent if $\mathcal{T}'$ is consistent. Thus, theory interpretations provide a mechanism for showing that one theory is consistent relative to another. One consequence of this is that IMPS can be used as a *foundational system* in which the user is allowed to only use theories which are known to be consistent relative to a chosen foundational theory (such as perhaps our theory of real numbers, `h-o-real-arithmetic`, which is described in the next subsection).

3.3 Example Theories

The most important theory in IMPS is a theory of higher-order real arithmetic called `h-o-real-arithmetic`. The theory contains a specification of the real numbers as a complete ordered field; the rational numbers and integers are specified as the usual substructures of the real numbers. The completeness axiom is formulated as a second-order sentence, which in the TEX output of IMPS has the form:

for every $p : [\mathbf{R}, prop]$ implication
- conjunction
  - $\exists\beta : \mathbf{R} \quad p(\beta)$
  - $\exists\alpha : \mathbf{R} \quad \forall\theta : \mathbf{R} \quad p(\theta) \supset \theta \leq \alpha$
- for some $\gamma : \mathbf{R}$ conjunction
  - $\forall\theta : \mathbf{R} \quad p(\theta) \supset \theta \leq \gamma$
  - $\forall\gamma_1 : \mathbf{R} \quad (\forall\theta : \mathbf{R} \quad p(\theta) \supset \theta \leq \gamma_1) \supset \gamma \leq \gamma_1.$

The theory `h-o-real-arithmetic` is equipped with routines for simplifying arithmetic expressions as well as rational linear inequalities (see Subsection 4.5). These routines allow the system to perform a great deal of low-level reasoning automatically. The theory contains several defined entities; e.g., the natural numbers are a defined sort and the higher-order operators Σ and Π are defined recursively.

As an encoding of the real numbers, `h-o-real-arithmetic` is an extremely useful theory building block. If a theory has `h-o-real-arithmetic` as a subtheory, the theory can be developed with the help of a large portion of basic, everyday mathematics. For example, in a theory of graphs with real arithmetic, one could introduce the very valuable concept of a weighted graph in which nodes or edges are assigned real numbers. We imagine that `h-o-real-arithmetic` will be a subtheory of almost every theory formulated in IMPS.

Several theories of abstract mathematical structures have been formulated in IMPS, including theories of monoids, groups, group actions, rings, and metric spaces. There is a family of "generic theories" for reasoning about quasi-constructors used to formulate objects such as sets, pairs, and sequences. These theories usually contain no nonlogical axioms (except for possibly the axioms of `h-o-real-arithmetic`); consequently, reasoning is performed in them using only the purely logical apparatus of LUTINS (and possibly real arithmetic). We have also developed various theories to support specific applications of IMPS in the area of software analysis, such as theories of state machines, abstract syntax, and denotational semantics.

Section 4
Theorem Proving

Theorem proving in IMPS is based on two levels of reasoning. Reasoning at the formula level is largely done automatically by the machine via an expression simplification routine. Reasoning at the proof structure level is done by user and the machine interactively. IMPS is designed to make great use of automated deduction without giving excessive reign to the machine; machine deduction is always orchestrated and controlled by the user.

IMPS produces formal proofs, but they are very different from the formal proofs that are described in logic text books. Usually a formal proof is a tree or graph constructed in a purely syntactic way from axioms, previously proved theorems, and a small number of low-level rules of inference. Formal proofs of this kind tend to be composed of a mass of small logical steps. It is no wonder that humans usually find these proofs to be unintelligible. In contrast, the steps in an IMPS proof can be very large, and most low-level inference in the proof is performed by the expression simplification routine. Since inference is described at a high-level, proofs constructed in IMPS resemble informal proofs, but unlike an informal proof, all the details of an IMPS proof are machine checked.

4.1 Deduction Graphs

Every proof is carried out within some formal theory. In the process of constructing a proof, IMPS builds a data structure representing the deduction, so that during the proof process the user has great freedom to decide the order in which he wants to work on different subgoals, and to try alternative strategies on a particular subgoal. At the end of a proof, this object, called a *deduction graph*, can be surveyed by the user or analyzed by software.

The items appearing in a deduction graph are not formulas, but *sequents*, in a sense derived from Gentzen [12]; see [24] for a discussion of the advantage of organizing deduction in this way. A sequent consists of a single formula called the *assertion* together with a *context*. The context is logically a set of assumptions, although the implementation caches various kinds of derived information with a context. In addition, the implementation associates each context with a particular theory. We will write a sequent in the form $\Gamma \Rightarrow A$, where Γ is a context and A is an assertion.

A deduction graph is a directed graph with nodes of two kinds, representing sequents and inferences respectively. If an arrow points from a sequent node to an inference node, then the sequent node represents a hypothesis to the inference. An inference node has exactly one arrow pointing at a sequent node, and that sequent node represents the conclusion of the inference. A sequent node is said to be *grounded* if at least one arrow comes into it from a grounded inference node; an inference node is grounded if, for every arrow coming into it, the source of the arrow is a grounded sequent node. In particular, an inference node with no arrows coming into it represents an inference with no hypotheses, and thus "closes" a path in the deduction graph. It is said to be "immediately grounded." A deduction graph may have one distinguished sequent node as its *goal*; it then represents the theorem to be proved.

This representation of deductions has several advantages. First, because any number of inference nodes may share a common sequent node as their conclusion, the user (or a program) may try any number of alternative strategies for proving a given sequent. Second, loops in deduction graphs arise naturally; they indicate that either of two sequents may be derived from the other, possibly in combination with different sets of additional premises. Finally, at the end of a proof, the resulting deduction graph serves as a transcript for analyzing the reasoning used in the proof, and recollecting the ideas.

4.2 Building Deduction Graphs

A deduction graph is begun by "posting" the goal node, a sequence node representing a sequent to be proved. The deduction graph is then enlarged by posting additional sequent nodes and creating inferences. The building of a deduction graph usually stops when the goal node is marked as grounded. Inference nodes are created by procedures called *primitive inferences*. Primitive inferences provide the only means to add inference nodes to a deduction graph; there is no way to modify or delete existing inference nodes. Each primitive inference works in roughly the same way: Certain information is fed to the primitive inference; zero or more new sequent nodes are posted; and finally, an inference node is constructed that links the newly posted nodes with one or more previously posted nodes.

There are about 30 primitive inferences. Two of the primitive inferences are special: `simplification` makes an inference on the basis of simplification (see Subsection 4.5); `macete-application` makes an inference by applying a macete (see Subsection 5.3). Each of the remaining primitive inferences embody one of the basic laws of LUTINS (or is a variant of `simplification`). For example, the primitive inference `direct-inference` applies an analogue of an introduction rule of Gentzen's sequent calculus (in reverse), according to the leading constructor of the assertion of the input sequent node. The system also has primitive inferences for beta-reduction, universal generalization, existential generalization, equality substitution, contraposition, cut, eliminating iota expressions, extensionality, unfolding defined constants, definedness assertions, defined-in-sort assertions, raising if-then-else expressions, recognizing tautologies, and for modifying the context of a sequent in various ways. Although the primitive inferences are available in every theory, some of them, such as `simplification` and `defined-constant-unfolding` depend on the axioms and theorems in the theory.

It is often inconvenient to call primitive inferences directly. For instance, `defined-constant-unfolding` takes, as one of its arguments, a set of paths to a defined constant that is to be unfolded. However, it can be quite difficult for a user to directly calculate the paths he is concerned with. This problem is addressed in IMPS by having a one or more *interface procedures* corresponding to each primitive inference. Each interface procedure (1) collects certain conveniently formulated information, (2) processes this information into a form appropriate for a particular primitive inference, and then (3) calls the primitive inference on the processed information. For example, corresponding to `defined-constant-unfolding` is an interface procedure that collects a set of natural numbers, where the number n represents the nth occurrence of the defined constant to be unfolded. The interface procedure calculates a path for each natural number and then calls `defined-constant-unfolding` with this new information.

for every $n : \mathbf{Z}$ implication
- $0 \leq n$
- $\sum_{j=0}^{n} j^6 = n^7/7 + n^6/2 + n^5/2 - n^3/6 + n/42$.

Figure 2: The Sum of Sixth Powers

for every $f, g : \mathbf{Z} \to \mathbf{R}$ implication
- for every $x : \mathbf{Z}$ $f(x) \leq g(x)$
- for every $m : \mathbf{Z}$ implication
  - $0 \leq m$
  - $\sum_{k=0}^{m} f(k) \leq \sum_{k=0}^{m} g(k)$.

Figure 3: The Monotonicity of Summation

4.3 Strategies

Strategies are procedures that call primitive inferences and interface procedures in useful patterns; they are akin to what are called tactics in some other systems, such as HOL [15], LCF [16], and Nuprl [5]. We have created a variety of strategies, both general and theory-specific. Some strategies facilitate the application of primitive inferences such as cut, equality substitution, and existential generalization.

An extremely important strategy is used for proving theorems by induction. The strategy takes, among other arguments, an *inductor* which specifies what induction principle to use, how to apply the induction principle, and what heuristics to employ in trying to prove the basis and induction step. IMPS allows the user to build his own inductors; the induction principles are axioms or theorems of an appropriate form. For example, the induction principle for the integers in `h-o-real-arithmetic` is just the full second-order induction axiom. The induction strategy is very effective on many theorems from elementary mathematics; in some simple cases, the strategy can produce a complete proof (two such formulas are printed in Figures 2–3), while in other cases it does part of the work and then returns control to the user. For instance, in the proof of the binomial theorem in commutative rings, the induction strategy proves the base case but does only a little processing on the induction step.

IMPS also has a family of "ending" strategies, the most basic of which is called `prove-by-logic-and-simplification`. These strategies correspond to statements like "and the theorem follows from the above lemmas" that are commonly given in informal proofs. They make complicated, but shallow inferences using lots of logical deduction and simplification. These strategies have the flavor of the proof search strategies of classic automated theorem provers; hence, they give IMPS a strong automated, as well as interactive, theorem proving capability.

4.4 Soundness

We intend, of course, that the user can only make sound inferences in IMPS. Our scheme for guaranteeing this is rather simple: IMPS allows the user to modify a deduction graph only by posting sequent nodes or by calling primitive inferences (either directly or indirectly). Since posting a sequent node does not effect the inferences encoded in a deduction graph, IMPS will be sound as long as each primitive inference is sound. The primitive inferences have been carefully implemented so that there is a high degree of assurance that they do indeed only make sound inferences. With this scheme, there is no problem about the soundness of interface procedures and strategies since they ultimately only affect a deduction graph through the application of primitive inferences. Hence, our machinery of deduction graphs and primitive inferences makes a type discipline like ML's unnecessary for assuring that complex reasoning does not go awry.

4.5 Simplification

Expression simplification is performed by the procedure `context-simplify`. This procedure applies to a context Γ (in a theory $\mathcal{T}$) and an expression e (of any syntactic type). It uses both theory-specific and general methods to compute an expression e' such that $\mathcal{T}$ and Γ together entail that e and e' have the same denotation.[7] The algorithm traverses the expression recursively; as it traverses propositional connectives it does simplification with respect to a richer context. Thus, for instance, in simplifying an implication $A \supset B$, A may be assumed true in the "local context" relative to which B is simplified. Similarly, in simplifying the last conjunct C of a ternary conjunction $A \wedge$

[7]That is, if either e or e' is defined, then both are, and in that case their values are equal.

$B \wedge C$, A and B may be assumed in the "local context." This strategy is justified in [24].

The procedure **context-simplify** is organized according to the top-most constructor or quasi-constructor of the expression to be simplified. Each constructor and quasi-constructor has its own simplification routine. A few constructors have very special routines that make use of information embodied in the axioms and theorems of the context's theory. The routines for the constructors **is-defined** and **is-defined-in-sort** do definedness checking with the help of the theory's domain-range handler (see Subsection 5.1). Expressions are simplified with the help of a theory-specific table of procedures. These procedures include rewrite rules, but in addition, certain algebraic theories (including **h-o-real-arithmetic**) make use of special-purpose simplification routines. These routines have been designed so that the simplification is done in a language independent way. Another built-in component of the simplifier is a decision procedure for rational linear inequalities. This component is also implemented in a language independent way.

A procedure called **context-entails?** uses **context-simplify** to check whether a formula is implied by a context. More precisely, **context-entails?** is a predicate that applies to a context Γ and a formula A which returns the Lisp value *true* if **context-simplify** can reduce A, relative to Γ, to the formula called **truth**. Intuitively, it tests whether the sequent $\Gamma \Rightarrow A$ is recognizable as valid using only trivial reasoning.

The procedures **context-simplify** and **context-entails?** are used systematically in the course of building deduction graphs. For instance, if Γ and A satisfy context entailment, then the sequent $\Gamma \Rightarrow A$ is considered immediately valid without any further inference. In addition, many kinds of inference are invariant with respect to context simplification. Hence, if A and B simplify to the same form relative to Γ, then the sequent $\Gamma \Rightarrow A$ can be replaced (in many positions) by $\Gamma \Rightarrow B$ without affecting the integrity of the deduction graph. This gives our proofs a degree of independence from the specific syntactic forms of the expressions occurring in them.

Since functions may be partial and terms may be undefined, term simplification in LUTINS must involve a certain amount of definedness checking. For example, simplifying expressions naively may cancel undefined terms, reducing an undefined expression such as $1/x - 1/x$ to 0 which is defined. In this example, the previous reduction is valid if the context Γ can be seen to entail the definedness or "convergence" of $1/x$. In general, algebraic reductions of this kind produce a certain number of intermediate definedness formulas which have to be considered by the simplifier. These formulas are called *convergence requirements.*

Despite these apparently stringent restrictions, the IMPS simplifier is able to work effectively. Although allowing partial functions in theories does introduce difficult problems in keeping track of and checking definedness of expressions, one of the significant lessons that we have learned from IMPS is that these difficulties can be overcome.

4.6 Proof Presentation

We are currently developing procedures for presenting proofs that have been constructed in IMPS. These procedures manipulate either the command history of a deduction graph or the deduction graph itself with the intention of highlighting the key steps while suppressing uninteresting details. The idea is that once a user has created a proof he should be able to create a presentation of the proof that is appropriate for a particular audience. The user can thus construct machine-checked proofs that are just as readable as ordinary informal proofs.

One basic procedure prints on the screen a full description of a given deduction graph in TeX. The procedure creates a TeX file that can also be used to print out the proof presentation on paper. Another useful procedure gives a more prescriptive TeX presentation of a deduction graph by presenting the deduction graph in terms of the commands (interface procedures and strategies) that were used to construct it. In Section 8, a proof of the Archimedean property of the real numbers is presented in TeX using both of these proof presentation procedures.

Section 5
Theory-Supported Reasoning

The logical content of a theory is determined by its language and set of axioms. As an IMPS object, a theory also has a variety of other characteristics, such as the sequence of defined constants that have been introduced, and the sequence of theorems that have been derived so far. This section will discuss three mechanisms that support theory-specific reasoning, by which we mean reasoning that is sound only relative to the axiomatic content of particular theories.

5.1 Reasoning about Definedness

Because LUTINS contains partial functions, it is important to automate, to greatest extent possible, the process of checking that expressions are well-defined or defined with a value in a particular sort. This kind of reasoning must rely heavily on axioms and theorems of the axiomatic theory at issue. The domain-range handler for a theory stores two primary kinds of information about the domain and range of function symbols in the language of the theory.

- *value* information: If a theorem is of the form

  $$\forall x_1 : \alpha_1, \ldots, x_n : \alpha_n \,.\, \phi(x_1, \ldots, x_n, f(x_1, \ldots, x_n))$$

  then it characterizes the range of f, and can be used in checking the definedness of expressions of the form $g(\ldots f(t_1, \ldots, t_n) \ldots)$.

- *definedness* information: A sufficient condition for the definedness of terms involving g is given by a theorem of the form

  $$\forall x_1 : \alpha_1, \ldots, x_n : \alpha_n \,.\, \psi(x_1, \ldots, x_n) \supset g(x_1, \ldots, x_n)\downarrow .$$

In addition, the domain-range handler keeps a list of *everywhere-defined* function symbols. For instance, of the arithmetic operators (considered on the reals), addition, subtraction, and multiplication are everywhere-defined; only division and exponentiation require information on definedness.

These facts are used in IMPS by an algorithm for checking definedness. Let Γ be the relevant context, and let t be the term in question. First, a variety of simple tests for definedness (using Γ) are applied. If they do not succeed, but t is of the form $g(t_1, \ldots, t_n)$, and we can (recursively) establish that $t_1, \ldots, t_n$ are all well-defined, then we consult the domain-range handler.

If g is known to be everywhere-defined, then t is defined. Otherwise, if g has definedness condition $\psi(x_1, \ldots, x_n)$, we form the new goal $\psi(t_1, \ldots, t_n)$. Moreover, for each subterm t_i that is of the form $f(t'_1, \ldots, t'_m)$ and has a value condition ϕ, we add $\phi(t'_1, \ldots, t'_m, f(t'_1, \ldots, t'_m))$ to Γ, thus forming an expanded context Γ'. Finally, we call `context-entails?` on Γ' and $\psi(t_1, \ldots, t_n)$.

A similar algorithm is used to check whether an expression is defined with a value in a particular sort, using another kind of information also maintained in the domain-range handler. This is *sort-definedness* information. A formula of the form

$$\forall x_1 : \alpha_1, \ldots, x_n : \alpha_n \,.\, \psi(x_1, \ldots, x_n) \supset (g(x_1, \ldots, x_n) \downarrow \alpha)$$

is used as a sufficient condition for g to be defined with a value in a particular sort α.

5.2 Transforms

Each theory contains a table with information used by the simplifier. This table is organized as a hash table of procedures (called transforms) each of which will transform an expression in a sound manner. Look-up in this table is done by using constructor and first lead constant as keys. Rewrite rules are implemented in this way, as are algebraic simplification procedures that would be impractical to represent as rewrite rules.

In IMPS some of the transforms can be generated in a uniform way, independently of the specific constants which play the role of the algebraic operations. This means that the simplifier can be crafted to provide particular forms of simplification, when the constants have certain algebraic properties. For instance, algebraic simplification for real arithmetic and for modular arithmetic are derived from the same entity, called an algebraic processor. An algebraic processor is applied by establishing a correspondence between the operators of the processor (e.g., the addition and multiplication operators) and specific constants of the theory. Certain operators need not be used; for instance, modular arithmetic in general does not have a division operator. Depending on the correspondence between operators and constants, the algebraic processor generates a set of formulas that must be theorems in the theory in order for its manipulations to be correct.

5.3 Macetes

In IMPS we have used the name *macete* (in Portuguese, a macete is a clever trick) to dcnote user-definable extensions of the simplifier which are under direct control of the user. They operate at a lower level than what we call strategies (see Subsection 4.3), but share an affinity to what are called tactics in some other systems. Formally, a macete is a function which takes as arguments a context and an expression and returns an expression. Macetes are used to apply a theorem or a collection of theorems to a sequent in a deduction graph. Individual theorems are applied by *theorem macetes* built automatically when a theorem is installed in a theory. Compound macetes are constructed from theorem macetes, some special macetes such as `beta-reduce` and `simplify`, and other compound macetes using a few simple *macete constructors*, which are just functions from macetes to macetes. This provides a simple mechanism for applying lists of theorems in a manner which is under direct user control.

One kind of theorem macete based on straightforward matching of expressions is called an *elementary macete.* An expression e matches a pattern expression p if and only if there is a substitution σ such that σ applied to p is α-equivalent to e. Though any kind of theorem can be used to generate an elementary macete, for the purposes of this exposition, let us assume the theorem is the universal closure of a conditional equality of the form $s \supset p_1 = p_2$. When applied to a context-expression pair (C, e), the macete works as follows. The left-hand side p_1 is matched to e; if this matching fails, then the macete simply returns e. If the matching succeeds, then the resulting substitution σ is applied to the formula s. If the resulting formula is entailed by the context C, then the macete returns the result of applying the substitution σ to the right-hand side p_2 of the original theorem. (This mechanism is described in more detail in [34].) Elementary macetes are used to apply a theorem within its home theory.

Another kind of theorem macete is called a *transportable macete.* It is based on a much more interesting kind of matching we call *translation match-*

ing, which allows for inter-theory matching of expressions. A translation match is essentially a two-fold operation consisting of a theory interpretation and ordinary matching. An expression e is a translation match to a pattern expression p if and only if there is a theory interpretation Φ and a substitution σ such that σ applied to the translation of p under Φ is α-equivalent to e. Apart from using translation matching instead of ordinary matching, transportable macetes work in much the same way as elementary macetes. Transportable macetes are used to apply a theorem outside of its home theory.

Section 6
Applications

The development of IMPS is currently being directed toward two application areas: mathematical analysis and software verification.

6.1 Mathematical Analysis

The development of IMPS has been guided, in large part, by our attempts to prove theorems in mathematical analysis—both theorems about the real numbers and theorems about more abstract objects such as continuous functions from one metric space to another. Mathematical analysis has traditionally served as a ground for testing the adequacy of formalizations of mathematics, because analysis requires great expressive power for constructing proofs. Nonetheless, surprisingly little has been done in the way of applying automated deduction to analysis (see Bledsoe's discussion [2]).

With partial functions, higher-order operators, and subtypes, LUTINS is well-suited as language for analysis. The value of having a natural way of dealing with partial functions in the development of analysis cannot be overestimated. Partial functions abound in analysis (as they do in most areas of mathematics), and many elegant theorems of analysis completely lose their elegance when they are expressed in a language having only total functions. Moreover, many of the important operators of analysis, such as the integral of a function and the limit of a sequence, are higher-order *partial* functions.

We have proved a variety of results leading up to a proof of the binomial theorem in commutative rings, including the combinatorial identity and various facts about Σ and Π. We have also proved in a theory of two metric spaces that the image of a connected set under a continuous mapping is itself connected (see [11]). These proofs are noteworthy because they correspond closely to standard proofs and because they are constructed by calling only a small number of commands.

6.2 Software Verification

We also believe that IMPS is well-suited to certain kinds of software verification. For instance, some approaches to specifying and verifying concurrent programs make use of traces or acceptance trees [20, 19]. These sequence-like or tree-like objects are easily formalized as partial functions on appropriate domains, and the operations and predicates used by the semantics can then be formalized as objects of the next higher type.

In addition, we have designed IMPS to be suited for reasoning about denotational definitions of programming languages. The standard approach to denotational semantics [33, 29] is rife with objects of higher type and expressions built using λ. Hence, a logic based on simple type theory seems highly appropriate to mechanizing reasoning in this area.

We are currently using IMPS on compiler verification for the Scheme programming language [27], which has a semantic definition in the denotational style. We have developed a theory of abstract syntax for Scheme and the target language of the compiler, together with a theory of the domains used in the denotational definitions of the two languages. Other applications are also underway.

Section 7
Implementation Notes

The IMPS program is written in T [28, 22], a sophisticated version of Scheme. The user interface is implemented using the subordinate process mechanism of GNU Emacs [32], which allows a program executing in T to issue commands to Emacs, and *vice versa.* Thus IMPS can request that formulas and derivations be presented to the user, specially formatted by Emacs, while conversely the user can frame his requests to IMPS using the interactive machinery of Emacs.

7.1 Syntax and Expressions

IMPS distinguishes between multiple user-oriented syntaxes, a basic s-expression syntax, and the logical expression itself. The logical expression is a T object with a great deal of cached information; logical expressions are uniquely instantiated in the system in the sense that the same abstract logical expression is never represented in two chunks of memory.[8] Translation between logical expressions and the s-expression syntax has a simple recursive character. Only this s-expression syntax is considered a basic part of IMPS. The s-expression syntax is then used as a basis to translate into various forms that are more appealing to users. For instance, we frequently use a string form akin to MACSYMA's representation of formulas; to inspect complex formulas we generate TEX code from the s-expression form and preview the results.

[8]Many other entities, such as contexts and sequents, are also uniquely instantiated in this sense.

Section 8
Example: the Archimedean Property

Ideally, we would like to give an example of *how* a proof is developed in IMPS, but unfortunately the "look and feel" of the IMPS user interface is quite difficult to capture in a conventional report such as this. Instead, we will present an example of the final product of the IMPS proof process. More specifically, we will take a deduction graph which proves that the real numbers satisfy the Archimedean property in the theory `h-o-real-arithmetic` and display it in TEX using IMPS proof presentation procedures.

We are interested in the following formula

$$\forall a : \mathbf{R} \, . \, \exists n : \mathbf{Z} \, . \, a < n,$$

which says that the real numbers are Archimedean, i.e., that every real number is dominated by some integer. The standard proof of this formula is by contradiction: Assume the negation of the formula; that is, assume there is some real number a greater than every integer. Hence, a is an upper bound for the set of integers, and so, by the completeness axiom of the real numbers, the integers have a least upper bound. However, since the integers are closed under addition of 1, the set of upper bounds of the integers must be closed under subtraction of 1, which contradicts the existence of a least upper bound of the integers.

This informal proof sketch can be straightforwardly formulated in IMPS by a deduction graph consisting of 22 nodes. Below are two TEX presentations of the proof contained in the deduction graph—one prescriptive and the other descriptive. The prescriptive presentation is given in terms of the commands (interface procedures and strategies) used to construct the deduction graph, while the descriptive presentation shows the full structure of the deduction graph. Both presentations were automatically generated from the deduction graph. The comments within square brackets have been added by hand.

It is important to keep in mind that this example illustrates only a very small part of the IMPS theorem proving mechanism. In fact, only seven different commands and nine primitive inferences were used in the construction of the deduction graph, and macetes were not used at all.

8.1 Prescriptive Presentation

Theorem

for every $a : \mathbf{R}$ for some $n : \mathbf{Z}$ $a < n$.

PROOF: Apply the strategy INSTANTIATE-THEOREM to the claim of the theorem. [Instantiate the completeness axiom of **h-o-real-arithmetic** with the predicate $\lambda\{x : \mathbf{Z} \mid \mathit{truth}\}$, and then take the result as an assumption.] This yields the following new subgoal:

Sequent 2.

Assume:

implication
- conjunction
  - $\mathit{nonvacuous?}\{\lambda\{x : \mathbf{Z} \mid \mathit{truth}\}\}$
  - $\exists \alpha : \mathbf{R} \;\; \forall \theta : \mathbf{R} \;\; \lambda\{x : \mathbf{Z} \mid \mathit{truth}\} \;\; (\theta) \supset \theta \leq \alpha$
- for some $\gamma : \mathbf{R}$ conjunction
  - $\forall \theta : \mathbf{R} \;\; \lambda\{x : \mathbf{Z} \mid \mathit{truth}\} \;\; (\theta) \supset \theta \leq \gamma$
  - $\forall \gamma_1 : \mathbf{R} \;\; (\forall \theta : \mathbf{R} \;\; \lambda\{x : \mathbf{Z} \mid \mathit{truth}\} \;\; (\theta) \supset \theta \leq \gamma_1) \supset \gamma \leq \gamma_1$.

Then $\forall a : \mathbf{R} \;\; \exists n : \mathbf{Z} \;\; a < n$.

Apply the interface procedure CONTRAPOSITION to the previous sequent. This [sets up a proof by contradication and] yields the following new subgoal:

Sequent 7.

Assume $\exists a : \mathbf{R} \;\; \forall n : \mathbf{Z} \;\; \mathit{not}(a < n)$. Then:

conjunction
- for some $\alpha : \mathbf{R}$ for every $\theta : \mathbf{R}$ implication
  - $\lambda\{x : \mathbf{Z} \mid \mathit{truth}\} \;\; (\theta)$
  - $\theta \leq \alpha$
- $\mathit{nonvacuous?}\{\lambda\{x : \mathbf{Z} \mid \mathit{truth}\}\}$
- for every $\gamma : \mathbf{R}$ disjunction
  - $\exists \theta : \mathbf{R} \;\; \lambda\{x : \mathbf{Z} \mid \mathit{truth}\} \;\; (\theta) \wedge \mathit{not}(\theta \leq \gamma)$
  - $\exists \gamma_1 : \mathbf{R} \;\; (\forall \theta : \mathbf{R} \;\; \lambda\{x : \mathbf{Z} \mid \mathit{truth}\} \;\; (\theta) \supset \theta \leq \gamma_1) \wedge \mathit{not}(\gamma \leq \gamma_1)$.

Apply the interface procedure ANTECEDENT-INFERENCE to the previous sequent [in order to fix an a satisfying the assumption of the sequent]. This yields the following new subgoal:

Sequent 8.

The conclusion of sequent 7 holds, provided $\forall n : \mathbf{Z} \quad not(a < n)$.

Apply the interface procedure SIMPLIFICATION to the previous sequent. This yields the following new subgoal:

Sequent 9.

Under the same assumptions as sequent 8, we have:

conjunction

- for some $\alpha : \mathbf{R}$ for every $\theta : \mathbf{R}$ implication
  - $\theta \downarrow \mathbf{Z}$
  - $\theta \leq \alpha$
- for every $\gamma : \mathbf{R}$ disjunction
  - $\exists \theta : \mathbf{R} \quad \theta \downarrow \mathbf{Z} \wedge not(\theta \leq \gamma)$
  - $\exists \gamma_1 : \mathbf{R} \quad (\forall \theta : \mathbf{R} \quad \theta \downarrow \mathbf{Z} \supset \theta \leq \gamma_1) \wedge not(\gamma \leq \gamma_1)$.

Apply the strategy DIRECT-INFERENCE-STRATEGY to the previous sequent. This yields the following new subgoals:

Sequent 10.

Under the same assumptions as sequent 8, we have:

for some $\alpha : \mathbf{R}$ for every $\theta : \mathbf{R}$ implication

- $\theta \downarrow \mathbf{Z}$
- $\theta \leq \alpha$.

Sequent 13.

Assume:

0. for every $\theta : \mathbf{R}$ implication
   - $\theta \downarrow \mathbf{Z}$
   - $\theta \leq \gamma$.
1. for some $\alpha : \mathbf{R}$ for every $\theta : \mathbf{R}$ implication
   - $\theta \downarrow \mathbf{Z}$
   - $\theta \leq \alpha$.
2. for every $n : \mathbf{Z}$ $not(a < n)$.

Then:

for some $\gamma_1 : \mathbf{R}$ conjunction
- for every $\theta : \mathbf{R}$ implication
  - $\theta \downarrow \mathbf{Z}$
  - $\theta \leq \gamma_1$
- $not(\gamma \leq \gamma_1)$.

Apply the strategy INSTANTIATE-EXISTENTIAL to the sequent 10. [Instantiate α with a.] This yields the following new subgoal:

Sequent 14.

Under the same assumptions as sequent 8, we have:

for every $\theta : \mathbf{R}$ implication
- $\theta \downarrow \mathbf{Z}$
- $\theta \leq a$.

Apply the interface procedure SIMPLIFICATION to the previous sequent [which immediately grounds the sequent]. Next, use the strategy INSTANTIATE-EXISTENTIAL applied to the sequent 13. [Instantiate γ_1 with $\gamma - 1$.] This yields the following new subgoal:

Sequent 16.

Under the same assumptions as sequent 13, we have:

conjunction
- for every $\theta_0 : \mathbf{R}$ implication
  - $\theta_0 \downarrow \mathbf{Z}$
  - $\theta_0 \leq \gamma - 1$
- $not(\gamma \leq \gamma - 1)$.

Apply the [ending] strategy PROVE-BY-LOGIC-AND-SIMPLIFICATION to the previous sequent [which grounds the sequent]. This completes the proof.

8.2 Descriptive Presentation

Theorem

for every $a : \mathbf{R}$ for some $n : \mathbf{Z}$ $a < n$.

PROOF: Inference based on the primitive inference CUT reduces sequent 1 to the subgoal sequents 2 and 3. [Note: The assumption of sequent 2 is the completeness axiom of `h-o-real-arithmetic` instantiated with the predicate $\lambda\{x : \mathbf{Z} \mid truth\}$.]

Sequent 2.

Assume:

implication
- conjunction
  - $nonvacuous?\{\lambda\{x : \mathbf{Z} \mid truth\}\}$
  - $\exists \alpha : \mathbf{R} \quad \forall \theta : \mathbf{R} \quad \lambda\{x : \mathbf{Z} \mid truth\} \quad (\theta) \supset \theta \leq \alpha$
- for some $\gamma : \mathbf{R}$ conjunction
  - $\forall \theta : \mathbf{R} \quad \lambda\{x : \mathbf{Z} \mid truth\} \quad (\theta) \supset \theta \leq \gamma$
  - $\forall \gamma_1 : \mathbf{R} \quad (\forall \theta : \mathbf{R} \quad \lambda\{x : \mathbf{Z} \mid truth\} \quad (\theta) \supset \theta \leq \gamma_1) \supset \gamma \leq \gamma_1$.

Then $\forall a : \mathbf{R} \quad \exists n : \mathbf{Z} \quad a < n$.

Sequent 3.

implication

- conjunction
  - ∘ $nonvacuous?\{\lambda\{x:\mathbf{Z}\,|\,truth\}\}$
  - ∘ $\exists\alpha:\mathbf{R}\quad\forall\theta:\mathbf{R}\quad\lambda\{x:\mathbf{Z}\,|\,truth\}\quad(\theta)\supset\theta\leq\alpha$
- for some $\gamma:\mathbf{R}$ conjunction
  - ∘ $\forall\theta:\mathbf{R}\quad\lambda\{x:\mathbf{Z}\,|\,truth\}\quad(\theta)\supset\theta\leq\gamma$
  - ∘ $\forall\gamma_1:\mathbf{R}\quad(\forall\theta:\mathbf{R}\quad\lambda\{x:\mathbf{Z}\,|\,truth\}\quad(\theta)\supset\theta\leq\gamma_1)\supset\gamma\leq\gamma_1.$

Inference based on the primitive inference CONTRAPOSITION reduces sequent 2 to

Sequent 7.

Assume $\exists a:\mathbf{R}\quad\forall n:\mathbf{Z}\quad not(a<n)$. Then:

conjunction

- for some $\alpha:\mathbf{R}$ for every $\theta:\mathbf{R}$ implication
  - ∘ $\lambda\{x:\mathbf{Z}\,|\,truth\}\quad(\theta)$
  - ∘ $\theta\leq\alpha$
- $nonvacuous?\{\lambda\{x:\mathbf{Z}\,|\,truth\}\}$
- for every $\gamma:\mathbf{R}$ disjunction
  - ∘ $\exists\theta:\mathbf{R}\quad\lambda\{x:\mathbf{Z}\,|\,truth\}\quad(\theta)\wedge not(\theta\leq\gamma)$
  - ∘ $\exists\gamma_1:\mathbf{R}\quad(\forall\theta:\mathbf{R}\quad\lambda\{x:\mathbf{Z}\,|\,truth\}\quad(\theta)\supset\theta\leq\gamma_1)\wedge not(\gamma\leq\gamma_1).$

Inference based on the primitive inference FOR-SOME-ANTECEDENT-INFERENCE reduces sequent 7 to

Sequent 8.

The conclusion of sequent 7 holds, provided $\forall n:\mathbf{Z}\quad not(a<n)$.

Inference based on the primitive inference SIMPLIFICATION reduces sequent 8 to

Sequent 9.

Under the same assumptions as sequent 8, we have:

conjunction

- for some $\alpha : \mathbf{R}$ for every $\theta : \mathbf{R}$ implication
  - $\theta \downarrow \mathbf{Z}$
  - $\theta \leq \alpha$
- for every $\gamma : \mathbf{R}$ disjunction
  - $\exists \theta : \mathbf{R} \quad \theta \downarrow \mathbf{Z} \wedge not(\theta \leq \gamma)$
  - $\exists \gamma_1 : \mathbf{R} \quad (\forall \theta : \mathbf{R} \quad \theta \downarrow \mathbf{Z} \supset \theta \leq \gamma_1) \wedge not(\gamma \leq \gamma_1)$.

Inference based on the primitive inference CONJUNCTION-DIRECT-INFERENCE reduces sequent 9 to the subgoal sequents 10 and 11.

Sequent 10.

Under the same assumptions as sequent 8, we have:

for some $\alpha : \mathbf{R}$ for every $\theta : \mathbf{R}$ implication

- $\theta \downarrow \mathbf{Z}$
- $\theta \leq \alpha$.

Sequent 11.

Assume:

0. for some $\alpha : \mathbf{R}$ for every $\theta : \mathbf{R}$ implication
   - $\theta \downarrow \mathbf{Z}$
   - $\theta \leq \alpha$.
1. for every $n : \mathbf{Z}$ $not(a < n)$.

Then:

for every $\gamma : \mathbf{R}$ disjunction
- for some $\theta : \mathbf{R}$ conjunction
  - $\theta \downarrow \mathbf{Z}$
  - $not(\theta \leq \gamma)$
- for some $\gamma_1 : \mathbf{R}$ conjunction
  - $\forall \theta : \mathbf{R} \quad \theta \downarrow \mathbf{Z} \supset \theta \leq \gamma_1$
  - $not(\gamma \leq \gamma_1)$.

Inference based on the primitive inference EXISTENTIAL-GENERALIZATION reduces sequent 10 to the subgoal sequents 14 and 15.

Sequent 14.

Under the same assumptions as sequent 8, we have:

for every $\theta : \mathbf{R}$ implication
- $\theta \downarrow \mathbf{Z}$
- $\theta \leq a$.

Note that this sequent is immediately grounded by SIMPLIFICATION.

Sequent 15.

Under the same assumptions as sequent 8, we have:

$a \downarrow \mathbf{R}$.

Note that this sequent is immediately grounded by SIMPLIFICATION.

Inference based on the primitive inference FOR-ALL-DIRECT-INFERENCE reduces sequent 11 to

Sequent 12.

Under the same assumptions as sequent 11, we have:

disjunction
- for some $\theta : \mathbf{R}$ conjunction
  - $\theta \downarrow \mathbf{Z}$
  - $not(\theta \leq \gamma)$
- for some $\gamma_1 : \mathbf{R}$ conjunction
  - $\forall \theta : \mathbf{R} \quad \theta \downarrow \mathbf{Z} \supset \theta \leq \gamma_1$
  - $not(\gamma \leq \gamma_1)$.

Inference based on the primitive inference DISJUNCTION-DIRECT-INFERENCE reduces sequent 12 to

Sequent 13.

Assume:

0. for every $\theta : \mathbf{R}$ implication
   - $\theta \downarrow \mathbf{Z}$
   - $\theta \leq \gamma$.
1. for some $\alpha : \mathbf{R}$ for every $\theta : \mathbf{R}$ implication
   - $\theta \downarrow \mathbf{Z}$
   - $\theta \leq \alpha$.
2. for every $n : \mathbf{Z}$ $not(a < n)$.

Then:

for some $\gamma_1 : \mathbf{R}$ conjunction
- for every $\theta : \mathbf{R}$ implication
  - $\theta \downarrow \mathbf{Z}$
  - $\theta \leq \gamma_1$
- $not(\gamma \leq \gamma_1)$.

Inference based on the primitive inference EXISTENTIAL-GENERALIZATION reduces sequent 13 to the subgoal sequents 16 and 17.

Sequent 16.

Under the same assumptions as sequent 13, we have:

conjunction

- for every $\theta_0 : \mathbf{R}$ implication
  - ∘ $\theta_0 \downarrow \mathbf{Z}$
  - ∘ $\theta_0 \leq \gamma - 1$
- $not(\gamma \leq \gamma - 1)$.

Sequent 17.

Under the same assumptions as sequent 13, we have:

$\gamma - 1 \downarrow \mathbf{R}$.

Note that this sequent is immediately grounded by SIMPLIFICATION.

Inference based on the primitive inference SIMPLIFICATION reduces sequent 16 to

Sequent 18.

Under the same assumptions as sequent 13, we have:

for every $\theta_0 : \mathbf{R}$ implication

- $\theta_0 \downarrow \mathbf{Z}$
- $1 + \theta_0 \leq \gamma$.

Inference based on the primitive inference FOR-SOME-ANTECEDENT-INFERENCE reduces sequent 18 to

Sequent 19.

Assume:

0. for every $\theta : \mathbf{R}$ implication
   - $\theta \downarrow \mathbf{Z}$
   - $\theta \leq \alpha$.

1. for every $\theta : \mathbf{R}$ implication
   - $\theta \downarrow \mathbf{Z}$
   - $\theta \leq \gamma$.
2. for every $n : \mathbf{Z}$ $not(a < n)$.

Then:

for every $\theta_0 : \mathbf{R}$ implication
- $\theta_0 \downarrow \mathbf{Z}$
- $1 + \theta_0 \leq \gamma$.

Inference based on the primitive inference FOR-ALL-DIRECT-INFERENCE reduces sequent 19 to

Sequent 20.

Under the same assumptions as sequent 19, we have:

implication
- $\theta_0 \downarrow \mathbf{Z}$
- $1 + \theta_0 \leq \gamma$.

Inference based on the primitive inference IMPLICATION-DIRECT-INFERENCE reduces sequent 20 to

Sequent 21.

Assume:

0. $\theta_0 \downarrow \mathbf{Z}$.
1. for every $\theta : \mathbf{R}$ implication
   - $\theta \downarrow \mathbf{Z}$
   - $\theta \leq \alpha$.
2. for every $\theta : \mathbf{R}$ implication
   - $\theta \downarrow \mathbf{Z}$
   - $\theta \leq \gamma$.
3. for every $n : \mathbf{Z}$ $not(a < n)$.

Then:

$1 + \theta_0 \leq \gamma$.

Inference based on the primitive inference BACKCHAIN-INFERENCE reduces sequent 21 to

Sequent 22.

Under the same assumptions as sequent 21, we have:

$1 + \theta_0 \downarrow \mathbf{Z}$.

Note that this sequent is immediately grounded by SIMPLIFICATION.

Inference based on the primitive inference UNIVERSAL-INSTANTIATION reduces sequent 3 to the subgoal sequents 4 and 5.

Sequent 4.

for every $p : \mathbf{R} \rightarrow *$ implication
- conjunction
  - *nonvacuous?*$\{p\}$
  - $\exists \alpha : \mathbf{R} \quad \forall \theta : \mathbf{R} \quad p(\theta) \supset \theta \leq \alpha$
- for some $\gamma : \mathbf{R}$ conjunction
  - $\forall \theta : \mathbf{R} \quad p(\theta) \supset \theta \leq \gamma$
  - $\forall \gamma_1 : \mathbf{R} \quad (\forall \theta : \mathbf{R} \quad p(\theta) \supset \theta \leq \gamma_1) \supset \gamma \leq \gamma_1$.

Sequent 5.

$\lambda\{x : \mathbf{Z} \,|\, truth\} \downarrow [\mathbf{R}, *]$.

Note that this sequent is immediately grounded by SIMPLIFICATION.

Inference based on the primitive inference THEOREM-ASSUMPTION reduces sequent 4 to

Sequent 6.

Assume:

for every $p : \mathbf{R} \rightarrow *$ implication
- conjunction
  - *nonvacuous?*$\{p\}$
  - $\exists \alpha : \mathbf{R} \quad \forall \theta : \mathbf{R} \quad p(\theta) \supset \theta \leq \alpha$
- for some $\gamma : \mathbf{R}$ conjunction
  - $\forall \theta : \mathbf{R} \quad p(\theta) \supset \theta \leq \gamma$
  - $\forall \gamma_1 : \mathbf{R} \quad (\forall \theta : \mathbf{R} \quad p(\theta) \supset \theta \leq \gamma_1) \supset \gamma \leq \gamma_1$.

Then:

for every $p : \mathbf{R} \rightarrow *$ implication

- conjunction
  - $nonvacuous?\{p\}$
  - $\exists \alpha : \mathbf{R} \ \ \forall \theta : \mathbf{R} \ \ p(\theta) \supset \theta \leq \alpha$
- for some $\gamma : \mathbf{R}$ conjunction
  - $\forall \theta : \mathbf{R} \ \ p(\theta) \supset \theta \leq \gamma$
  - $\forall \gamma_1 : \mathbf{R} \ \ (\forall \theta : \mathbf{R} \ \ p(\theta) \supset \theta \leq \gamma_1) \supset \gamma \leq \gamma_1.$

Note that this sequent is immediately grounded by SIMPLIFICATION. [This completes the proof.]

Section 9
Conclusion

IMPS is an interactive proof development system intended to support standard mathematical notation, concepts, and techniques. In particular, it provides a flexible logical framework in which to specify axiomatic theories, prove theorems, and relate one theory to another via inclusion and theory interpretation. Theory interpretations, which are extremely useful for reusing theorems and theories, are used extensively in IMPS. The IMPS logic is a conceptually simple, but highly expressive version of higher-order logic which allows partially defined (higher-order) functions and undefined terms. The simple types hierarchy of the logic is equipped with a very effective subtyping mechanism. Proofs are developed in IMPS with the aid of several different deduction mechanisms, including expression simplification, automatic theorem application, and a user-extensible mechanism for orchestrating applications of inference rules and theorems. The naturalness of the logic and the high level of inference in proofs make it possible to develop machined-checked proofs in IMPS that are very intuitive and readable. Finally, the unique and congenial IMPS user interface enables the user to control and understand the deduction process, and to inspect and present proofs using TEX.

List Of References

1. P. B. ANDREWS, S. ISSAR, D. NESMITH, AND F. PFENNIG, *The* TPS *theorem proving system (system abstract)*, in 10th International Conference on Automated Deduction, M. E. Stickel, ed., vol. 449 of Lecture Notes in Computer Science, Springer-Verlag, 1990, pp. 641–642.

2. W. W. BLEDSOE, *Some automatic proofs in analysis*, in Automated Theorem Proving: After 25 Years, American Mathematical Society, 1984.

3. G. S. BOOLOS, *On second-order logic*, Journal of Philosophy, 72 (1975), pp. 509–527.

4. L. CARDELLI AND P. WEGNER, *On understanding types, data abstraction, and polymophism*, Computing Surveys, 17 (1985), pp. 471–522.

5. R. L. CONSTABLE, S. F. ALLEN, H. M. BROMLEY, W. R. CLEAVELAND, J. F. CREMER, R. W. HARPER, D. J. HOWE, T. B. KNOBLOCK, N. P. MENDLER, P. PANANGADEN, J. T. SASAKI, AND S. F. SMITH, *Implementing Mathematics with the Nuprl Proof Development System*, Prentice-Hall, Englewood Cliffs, New Jersey, 1986.

6. T. COQUAND AND G. HUET, *The calculus of constructions*, Information and Computation, 76 (1988), pp. 95–120.

7. H. B. ENDERTON, *A Mathematical Introduction to Logic*, Academic Press, 1972.

8. W. M. FARMER, *Abstract data types in many-sorted second-order logic*, Tech. Rep. M87-64, The MITRE Corporation, 1987.

9. ——, *A partial functions version of Church's simple theory of types*, Journal of Symbolic Logic, 55 (1990), pp. 1269–91.

10. ——, *A simple type theory with partial functions and subtypes.* Forthcoming.

11. W. M. FARMER AND F. J. THAYER, *Two computer-supported proofs in metric space topology.* Forthcoming.

12. G. GENTZEN, *Investigations into logical deduction (1935)*, in The Collected Works of Gerhard Gentzen, North Holland, 1969.

13. J. A. GOGUEN, *Reusing and interconnecting software components*, Computer, 10 (1986), pp. 528–543.

14. ——, *Principles of parameterized programming*, tech. rep., SRI International, 1987.

15. M. GORDON, HOL*: A proof-generating system for higher-order logic*, in VLSI Specification, Verification and Synthesis, Kluwer, 1987, pp. 73–128.

16. M. GORDON, R. MILNER, AND C. P. WADSWORTH, *Edinburgh* LCF*: A Mechanised Logic of Computation*, vol. 78 of Lecture Notes in Computer Science, Springer Verlag, 1979.

17. J. D. GUTTMAN, *A proposed interface logic for verification environments*, Tech. Rep. M91-19, The MITRE Corporation, 1991.

18. L. HENKIN, *Completeness in the theory of types*, Journal of Symbolic Logic, 15 (1950), pp. 81–91.

19. M. HENNESSY, *Algebraic Theory of Processes*, MIT Press, 1988.

20. C. A. R. HOARE, *Communicating Sequential Processes*, Prentice-Hall International, Englewood Cliffs, NJ, 1985.

21. W. A. HOWARD, *The formulae-as-types notion of construction*, in To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism, Academic Press, 1980, pp. 479–490.

22. D. KRANZ, R. KELSEY, J. REES, P. HUDAK, J. PHILBIN, AND N. ADAMS, ORBIT*: An optimizing compiler for scheme*, SIGPLAN Notices, 21 (1986), pp. 219–233. Proceedings of the '86 Symposium on Compiler Construction.

23. P. MARTIN-LÖF, *Constructive mathematics and computer programming*, in Logic, Methodology, and Philosophy of Science VI, L. J. Cohen, J. Los, H. Pfeiffer, and K. P. Podewski, eds., Amsterdam, 1982, North-Holland, pp. 153–175.

24. L. G. MONK, *Inferencc rules using local contexts*, Journal of Automated Reasoning, 4 (1988), pp. 445–462.

25. Y. N. MOSCHOVAKIS, *Elementary Induction on Abstract Structures*, North-Holland, 1974.

26. ——, *Abstract recursion as a foundation for the theory of algorithms*, in Computation and Proof Theory, Lecture Notes in Mathematics 1104, Springer-Verlag, 1984, pp. 289–364.

27. J. REES AND W. CLINGER EDS., *Revised^3 report on the algorithmic language scheme*, ACM SIGPLAN Notices, 21 (1986), pp. 37–79.

28. J. A. REES, N. I. ADAMS, AND J. R. MEEHAN, *The T Manual*, Computer Science Department, Yale University, 5th ed., 1988.

29. D. A. SCHMIDT, *Denotational Semantics: A Methodology for Language Development*, Wm. C. Brown, Dubuque, IO, 1986.

30. S. SHAPIRO, *Second-order languages and mathematical practice*, Journal of Symbolic Logic, 50 (1985), pp. 660–696.

31. J. R. SHOENFIELD, *Mathematical Logic*, Addison-Wesley, 1967.

32. R. M. STALLMAN, GNU *Emacs Manual (Version 18)*, Free Software Foundation, 6th ed., 1987.

33. J. E. STOY, *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory*, MIT Press, Cambridge, MA, 1977.

34. F. J. THAYER, *Obligated term replacements*, Tech. Rep. MTR-10301, The MITRE Corporation, 1987.